First Order Logic with Inductive Definitions for Model-Based Problem Solving

Maurice Bruynooghe
Department of Computer Science
KU Leuven
3001 Leuven, Belgium
Maurice.Bruynooghe@cs.kuleuven.be

Marc Denecker
Department of Computer Science
KU Leuven
3001 Leuven, Belgium
marc.denecker@cs.kuleuven.be

Miroslaw Truszczynski
Department of Computer Science
University of Kentucky
Lexington, KY 40506, USA
mirek@cs.uky.edu

Abstract

In ASP, programs can be viewed as specifications of finite Herbrand structures. Other logics can be (and, in fact, were) used towards the same end and can be taken as the basis of declarative programming systems of similar functionality as ASP. We discuss here one such logic, the logic FO(ID), and its implementation IDP3. The choice is motivated by notable similarities between ASP and FO(ID), even if both approaches trace back to different origins.

Introduction

Answer-set programming (ASP), described in other articles in the issue, is based on an extension of the language of logic programming under the answer-set semantics. In ASP, an instance of a computational problem is encoded by an answer set program in such a way that the Herbrand models of the program determine all solutions for the problem instance. Thus, at an abstract level, answer set programs are specifications of finite Herbrand structures (those that are models of the programs), and the key reasoning task supported by ASP systems is to compute them. That task is often referred to as *model generation*.

Other logics that can express constraints on Herbrand structures (or even non-Herbrand structures) could also be used as the basis for this form of declarative problem solving. Implementing the model generation task for theories in such logics yields declarative programming tools with the same basic functionality as that of ASP systems. Two examples of such an alternative approach are the model generators NP-SPEC (Cadoli et al. 2000) and *aspps* (East and Truszczynski 2006) for extended DATALOG. In both cases, the specification language is an extension of DATALOG with clausal constraint rules. This formalism captures the class NP in the sense that any decision problem from the class NP can be specified. The system NP-SPEC offers a compiler for recursion- and negation-free NP-SPEC problems to SAT whereas *aspps* is a native system supporting recursive DATALOG, making it suitable to model problems involving transitive closure.

A further step in this direction is the logic FO(ID). The origin of the logic FO(ID) can be traced to the fundamental problem of logic programming that also triggered the development of the stable semantics and later ASP, namely, the problem of negation as failure (cf. the paper *Answer Sets and the Language of Answer Set Programming* (Lifschitz 2016) in this issue). The essence of the negation problem was that negation as failure in Prolog had so many useful and intuitive applications, and yet, the original view of logic programs as sets of material implications (Horn clauses extended with negation in the body) could not account for the derivation of a single negative literal. As noted by Lifschitz in his paper in this issue, one solution was inspired by research in nonmonotonic reasoning and consisted of adapting the semantics of default logic (Reiter 1980) to the syntax of logic programs. This led to the stable semantics and later to ASP.

An alternative solution was to interpret a logic program as a *definition* of its predicates. In this view, a logic program consists of, for every predicate, an exhaustive list of rules defining the *cases* in which it is true. Unlike sets of implications, definitions entail negative information. Moreover, a case-based representation of definitions and, in particular, *inductive* definitions is a common way to specify definitions in mathematical texts. The view of logic programs as a definition was already implicit in Clark's first order completion semantics (Clark 1978). However, as was well-known in mathematical logic and databases, in general inductive definitions cannot be expressed in first order logic (FO) (Aho and Ullman 1979). Hence, Clark's semantics did not not correctly interpret recursive logic programs as inductive definitions. This weakness spurred the development of so-called *canonical semantics* such as the *perfect model* (Apt, Blair, and Walker 1988) and the *well-founded model* semantics (Van Gelder, Ross, and Schlipf 1991). The latter (in a suitably extended version) turned out to correctly formalize definitions, including the most common forms of inductive definitions (Denecker 1998; Denecker and Vennekens 2014). Since definitional knowledge is an important form of expert knowledge and since in general it cannot be expressed in FO, it is natural to seek extensions of classical logic to incorporate it. This has been recognized by the database community that developed such extensions in support of more expressive database query languages (Abiteboul, Hull, and Vianu 1995). It has also motivated a similar effort in *knowledge representation*. In particular, these considerations led

to the logic FO(ID), an extension of FO with inductive definitions expressed as sets of logic program-like rules under well-founded semantics that represent the base cases and (possibly) inductive cases of the definition (Denecker 2000; Denecker and Ternovska 2008).

The logic FO(ID) has a standard Tarskian model semantics. A structure satisfies a theory if it satisfies its FO sentences and is a well-founded model of its definitions. Thus, the logic FO(ID) can be understood as a conjunction of its formulas and definitions and, in particular, it is a conservative extension of first-order logic. These features make the logic FO(ID) well suited as the basis of declarative knowledge representation systems.

Two systems were developed for significant fragments of the FO(ID) language: *enfragmo* (Aavani et al. 2012), and IDP3 (De Cat et al. 2014). The key inference task supported by these systems is (finite) *model expansion*: given an FO(ID) theory and a partial structure as input, the goal is to output one or more structures that are models of the theory and expand the input structure. The *enfragmo* system provides support for arithmetic and aggregates in its theories, and also some limited support for inductive definitions. The IDP3 system has a similar functionality as *enfragmo* but provides a more advanced treatment and support for inductive definitions; in addition to model expansion, it also supports several other forms of inference.

In this paper, we present the logic FO(ID) and the IDP3 system. Our presentation is not formal but relies on a series of examples. We start by illustrating the importance of structures in knowledge representation and how they lead to a methodology for knowledge modeling and for declarative problem solving through the model expansion task. We then describe the IDP3 system. This discussion is intertwined with references to the logic FO(ID), the theoretical foundation for the IDP3 system. Next we briefly discuss the relationship between FO(ID) (IDP3) and ASP and conclude with comments on the role these formalisms may play in the future of computational logic.

Structures in Knowledge Representation

First-order logic has proved to be a powerful formalism for representing knowledge largely because of two key interrelated factors. Structures used as interpretations of first-order formulas are well suited to model practical situations and application domains; and our intuitive understanding of how first order formulas constrain the space of possible structures matches exactly the formal definition of the satisfiability relation. Our goal in this section is to present structures in their role as a fundamental abstraction for knowledge representation.

When modeling a problem domain, we start by selecting *symbols* to denote its functions and relations. Collectively, these symbols form the *vocabulary* of the domain. Each symbol in the vocabulary comes with a non-negative integer called the *arity* that denotes the number of arguments of the corresponding function or relation. If a is a relation or function symbol, we write a/k to indicate the arity of a. Function symbols of arity 0 are called *constants*.

To illustrate, let us consider a hypothetical software company that holds weekly lunch meetings for its software development teams. These lunch meetings take place on certain weekdays. Some teams are in a (scheduling) conflict (for instance, they may share a team member). The meeting days for teams in conflict must be different.

This short text describes a problem domain. The underlined terms indicate relations and functions in that domain. Figure 1 shows symbols that could be selected to denote them, as well as their intended meaning. Together, these symbols form the vocabulary of the *lunch meeting* domain.

Relation symbols
$team/1$, $day/1$, $conflict/2$
$team(x)$: x is a team
$day(x)$: x is a weekday
$conflict(x, y)$: x has a conflict with y

Function symbols (here only one)
$mtng\_day/1$
$mtng\_day(x) = y$: the time of lunch meeting of x is y

Figure 1: A vocabulary of symbols for the lunch meeting domain.

A *structure* (also called an *interpretation*) $\mathcal{S}$ over a vocabulary consists of a non-empty universe D and, for each symbol σ in the vocabulary, the *value* $\sigma^{\mathcal{S}}$ of σ in $\mathcal{S}$ (also called the interpretation of σ in $\mathcal{S}$). More specifically, for every relation symbol r/k, $r^{\mathcal{S}}$ is a relation on D with k arguments (that is, $r^{\mathcal{S}} \subseteq D^k$), and for every function symbol f/k, $f^{\mathcal{S}}$ is a function on D with k arguments (that is, $f^{\mathcal{S}} : D^k \to D$). Figure 2 shows an example of a structure over the vocabulary from Figure 1.

Domain of $\mathcal{S}$
$D^{\mathcal{S}} = \{T_1, T_2, T_3, T_4, T_5, M, Tu, W, Th, F\}$

Relations
$team^{\mathcal{S}} = \{T_1, T_2, T_3, T_4, T_5\}$
$day^{\mathcal{S}} = \{M, Tu, W, Th, F\}$
$conflict^{\mathcal{S}} =$
$\{(T_1, T_2), (T_1, T_5), (T_2, T_3), (T_2, T_5), (T_3, T_4)\}$

Functions
$mtng\_day^{\mathcal{S}} =$
$\{T_1 \to M, T_2 \to W, T_3 \to M, T_4 \to Tu, T_5 \to Tu,$
$M \to M, Tu \to M, W \to M, Th \to M, F \to M\}$

Figure 2: A structure $\mathcal{S}$ for the vocabulary from Figure 1.

A structure over a vocabulary is an abstract representation of a concrete instance, or *state of affairs*, of a problem domain modeled in terms of this vocabulary. The universe of the structure is an abstraction of the set of objects in that instance, while the relations and functions of the structure — the values of the symbols in the vocabulary — abstract the relations and functions in the instance.

For instance, the structure $\mathcal{S}$ in Figure 2 represents a state of affairs with five teams referred to as $T_1, \ldots, T_5$ and five

(working) days referred to as $M, \ldots, F$. The structure also specifies conflicts between the teams (e.g., teams T_1 and T_2 are in conflict), and an assignment of meeting days to teams (e.g., team T_3 meets on Monday). Since functions in structures are *total*, the function $mtng\_day^{\mathcal{S}}$ is also defined on days. This is redundant, as those assignments do not represent any pertinent information. The structure $\mathcal{S}$ is an abstraction of a *possible* state of affairs of our problem domain: one in which the properties of the domain mentioned in the specification of this problem domain are *satisfied*. Indeed, the mapping $mtng\_day^{\mathcal{S}}$ assigns weekdays to teams (other assignments it makes are immaterial or, as we said, redundant), and two teams in conflict are not scheduled to have lunch on the same day.

To formally model this domain, these and other properties present in the informal description (some only implicitly) need to be expressed as sentences of the logic over the chosen vocabulary. Three properties relevant to our scenario are shown in Figure 3. We see there formal sentences in the language of first-order logic over the vocabulary from Figure 1, as well as their informal reading. The first sentence says that the relation $conflict$ applies to teams only. It specifies the *types* of the arguments of the relation $conflict$. Incidentally, this information is not explicitly present in the narrative. Nevertheless, it is implicit there and can be included in any formal representation of the problem. The second sentence is of a similar nature. It describes the type of objects the function $mtng\_day$ maps to. The last sentence represents the *essential* constraint of the problem that teams in conflict do not hold their lunch meetings on the same day.

Vocabulary: As in Figure 1

Sentence:
$\forall X \forall Y (conflict(X, Y) \rightarrow team(X) \wedge team(Y))$

Reads: for all X and Y, if X and Y are in conflict then both X and Y are teams

Sentence: $\forall X \forall Y (mtng\_day(X) = Y \rightarrow day(Y))$

Reads: for all X and Y, if Y is the meeting day for X then Y is a day

Sentence: $\forall X \forall Y (conflict(X, Y) \rightarrow mtng\_day(X) \neq mtng\_day(Y))$

Reads: for all X and Y, if X and Y are in conflict then the meeting day for X is different from the meeting day for Y

Figure 3: Relevant properties as first-order sentences.

First-order propositions (FO sentences) are *true* or *false* in structures (we also say *satisfied* or *unsatisfied*, respectively). For example, the third proposition of Figure 3 expressing that conflicting teams do not meet on the same day is true in the structure from Figure 2. However, it is false in the structure that is the same except that the interpretation of $conflict$ is extended with the pair (T_4, T_5). Indeed, we now have two teams in conflict that are scheduled to meet on the same day.

More formally, given a structure $\mathcal{S}$ interpreting all symbols in a first-order sentence F, we can *evaluate* F in $\mathcal{S}$, that is, assign to it a logical value $true$ or $false$. When F evaluates to $true$ in $\mathcal{S}$, we say that $\mathcal{S}$ is a *model* of F or that F is *satisfied* by $\mathcal{S}$. The property of a sentence being true in a structure yields a *satisfaction relation* between structures and first-order logic sentences. It provides first-order logic sentences with a semantics (the *first-order semantics*) that captures precisely their informal reading. Figure 4 illustrates these concepts for sentences from the language based on our example vocabulary. The second sentence would evaluate to $false$ in case the relation $conflict^{\mathcal{S}}$ contained the extra pair (T_4, T_5).

Vocabulary: As in Figure 1
Structure: $\mathcal{S}$ defined in Figure 2

Sentence: $\forall X \forall Y (conflict(X, Y) \rightarrow team(X) \wedge team(Y))$
Evaluates to $true$. Indeed, for every $(a, b) \in conflict^{\mathcal{S}}$, both $team(a)$ and $team(b)$ hold in $\mathcal{S}$ (cf. Figure 2)

Sentence: $\forall X \forall Y (conflict(X, Y) \rightarrow mtng\_day(X) \neq mtng\_day(Y))$
Evaluates to $true$. Indeed, for every $(a, b) \in conflict^{\mathcal{S}}$, teams a and b meet on a different day (cf. Figure 2)

Figure 4: The first-order semantics applied to some sentences from Figure 3.

The satisfaction relation is of crucial importance. In some cases, we fully know the relevant state of affairs of the problem domain or, more precisely, we know the structure that serves as its abstract representation. However, in other cases the precise state of affairs is not known or is only partially known. In that case, our knowledge frequently consists of separate informal propositions. They implicitly specify *possible* states of affairs as those in which these propositions hold true. As we argued, structures are formal representations of states of affairs. Those structures that represent *possible* states of affairs are called *intended*. Given this terminology, knowledge representation can then be understood as the art and practice of formulating knowledge as a formal theory so that models of that theory are precisely the intended structures. For instance, the original specifications of the problem domain are correctly expressed by axioms in Figure 3, which also shows their informal semantics. The structure from Figure 2 is an intended one, and it indeed satisfies (is a model of) all the sentences in Figure 3. That latter claim can be verified formally and is also easy to see intuitively: conflicts are between teams, the mapping $mtng\_day^{\mathcal{S}}$ assigns days to teams (and also to days, but that is immaterial), and two teams in conflict are not scheduled on the same day.

To say that all models of the theory are intended structures here is slightly imprecise. For example, the theory has infinite models which hardly count as intended structures. The problem is that some implicit information such as what are weekdays and teams, is not expressed in the theory. This information is expressed in the values assigned to the symbols $team$ and day by structures like the one in Figure 2.

To recapitulate, in this setting the satisfaction relation allows us to use sentences over the fixed vocabulary to con-

strain structures over that vocabulary to those that satisfy the sentences. These sentences can be seen as *specifications* of classes of intended structures over that vocabulary, that is, the structures that represent those states of affairs that are possible (might be encountered in practice).

The setting we presented supports several important reasoning problems. Say the manager in our running example is reviewing a schedule proposed by one of her assistants or, more formally, the corresponding structure. The manager wants to know whether certain propositions hold for the schedule or, formally, whether the formal sentences expressing the propositions are satisfied in that structure. We call that reasoning task *model checking* or *querying*. For instance, we might want to know whether team T_2 has its meeting scheduled on the same day as team T_4 in the structure (schedule) in Figure 2 (that *query* would evaluate to $false$). *Model checking* is a special instance of this task; it verifies that a structure satisfies the specifications, that is, that it indeed is an intended structure. In the case of our example and the structure in Figure 2, it consists of verifying that all statements of the theory in Figure 3 are satisfied by the structure.

Even more interesting and important is the situation when the schedule is yet to be constructed. How can the manager find one? She knows the axioms in Figure 3. This information specifies the class of intended structures, each of them representing a valid instance of a lunch meeting domain. It is also reasonable to assume that she knows which teams she needs to schedule, what scheduling conflicts she has to take into account, and which days are work days. This information explicitly fixes the domain of an intended structure as well as its relations $team$, day and $conflict$ (for instance, to the values they have in Figure 2). Any function $mtng\_day$ that completes this explicitly given fragment of a structure to an intended one yields a good schedule for the setting of interest to the manager. The converse is also true. Good schedules give rise to intended structures (when combined with the explicitly given components).

The task to find the missing function, which we just described, is an example of the *model expansion* problem. In *model expansion* we assume that the vocabulary is partitioned into *input* and *output* symbols. Given a *theory* (that is, a set of sentences) over the entire vocabulary and a structure over the vocabulary consisting of the input symbols, called an *input* structure, the goal is to extend the input structure with relations and functions for output symbols so that the resulting structure (now over the entire vocabulary) satisfies the theory.

This applies to our example scenario. Here, $team$, day and $conflict$ are input symbols and $mtng\_day$ is an output symbol. The input structure consists of the domain $\{T_1, \ldots, T_5, M, \ldots, F\}$ and of the relations $team$, day, and $conflict$ as in Figure 2. The theory specifying intended structures is given in Figure 3. Under these assumptions, the model expansion problem asks for a specific function $mtng\_day$ that would expand the input structure to the one modeling the three sentences (that is, to an intended structure). That function would offer a legal schedule of lunch meetings for the five teams involved. The function shown in Figure 2 is one of possible solutions. The function in Figure 5 is another one.[1]

$$
\begin{array}{l}
mtng\_day = \\
\quad \{T_1 \to M, T_2 \to W, T_3 \to M, T_4 \to W, T_5 \to Tu, \\
\quad\ M \to M, Tu \to M, W \to M, Th \to M, F \to M\}
\end{array}
$$

Figure 5: Another possible schedule function.

A more involved reasoning task is assigning meeting days to teams *interactively*. The task involves *propagation* inference that calculates the valid days for every team. Each time the user assigns a meeting day to a team, the propagation inference updates the valid choices for the remaining teams. Still another task is *revision*, assisting a user in assigning a different meeting day to a particular team while preserving as much as possible the meeting days assigned to the other teams. All these reasoning tasks use the same theory as a specification of valid structures.

To recapitulate, structures are important to us for four key reasons. First, they provide natural abstractions of states of affairs of the problem domain, in which sentences (properties) can be evaluated for satisfaction. Second, they are useful to define computational tasks in the context of logic. Third, they can be used to present input data, as in the query inference (where the value of every symbol is known), or in model expansion (where values of some symbols are not known). Fourth, they can be used as representations of *answers* to model expansion problems.

The IDP3 System

We will now present a software system IDP3 that implements the ideas presented above. In particular, IDP3 allows us to define structures, input structures and partial structures, as well as sentences to state their properties.[2] An overview of the IDP3 system is presented in Figure 6. We use a series of simple examples to illustrate and discuss all key features of IDP3.

The IDP3 system separates information from the reasoning task to be performed. In this way, it facilitates the use of the same knowledge to solve diverse reasoning problems. To represent information, IDP3 uses an enriched variant of first order logic. The information is split over three components. The first component is the *vocabulary*. The IDP3 syntax for describing vocabularies is illustrated in Figure 7, where we again use our software teams domain as an example. The vocabulary goes beyond the basic first order logic

[1] A more general version of the model expansion problem takes a partially instantiated structure (a fully specified domain but possibly only partially instantiated relations and functions for all vocabulary symbols) and asks if it can be completed to a structure that satisfies the theory.

[2] The IDP3 system has been developed by the Knowledge Representation and Reasoning group at the University of Leuven `https://dtai.cs.kuleuven.be/topics/kbs`. The most recent versions of the source code and documentation, as well as other resources such as an on-line IDE are available at the IDP page `https://dtai.cs.kuleuven.be/software/idp`.

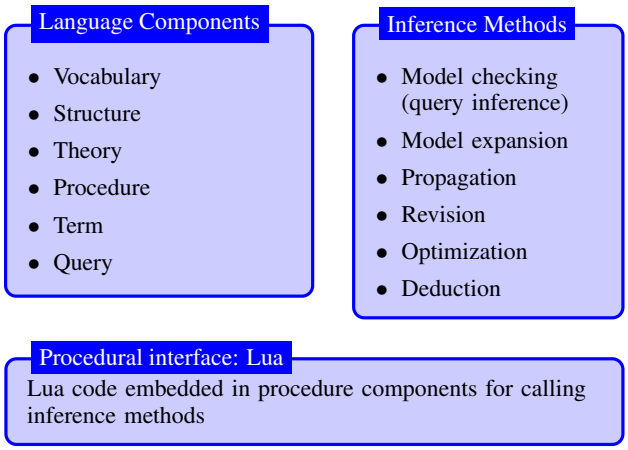

Figure 6: High-level representation of IDP3 as a Knowledge base System.

as it introduces not only the alphabet but also the *types* and the *signatures* of the relation and function symbols. In IDP3, vocabularies are assigned identifiers; the vocabulary in Figure 7 is identified by V.

```
vocabulary V{
    type day
    type team
    conflict(team,team)
    mtng_day(team):day
}
```

Figure 7: An IDP3 vocabulary introducing two types *day* and *team*, and a predicate and a function together with their signatures.

The second component is a *structure*. The IDP3 syntax for a structure for our example domain is shown in Figure 8. The structure has an identifier (here S). It refers to a vocabulary (here V) and introduces the domains for the types declared in the vocabulary (here by enumerating the elements of types *day* and *team*). In addition it also enumerates the known information about the declared relations and functions. The interpretation of *conflict* is fully known, and the relation is specified by the list of its tuples. On the other hand, as nothing is known about the interpretation of $mtng\_day$, nothing about this function is mentioned in the structure.

```
structure S:V{
    day={M;Tu;W;Th;F}
    team={T1;T2;T3;T4;T5}
    conflict={(T1,T2);(T1,T5);(T2,T3);(T2,T5);(T3,T4)}
}
```

Figure 8: An IDP3 structure for the vocabulary V from Figure 7.

The last component used to express information is a *theory*. An example theory appropriate for our domain is shown in Figure 10. The notation is essentially first order logic but in a keyboard-friendly syntax. The IDP3 counterparts to standard mathematical logic notation are given in Figure 9.

FO	IDP3	FO	IDP3	FO	IDP3
$\wedge$	&	$\equiv$	<=>	=	=
$\vee$	|	$\neg$	~	$\neq$	~=
$\rightarrow$	=>	$\forall$	!	$\leq$	=<
$\leftarrow$	<=	$\exists$	?	$\geq$	>=

Figure 9: Translation table. Note that implication translates into a double arrow, while the translation of $\leq$ and $\geq$ is not an arrow.

The theory (as the other two components) has an identifier (here T). The theory refers to a vocabulary (in the example, via the identifier V), and expresses the constraints that a structure must meet to serve as a valid abstraction of the problem domain (here, the function $mtng\_day$ is constrained so that teams in conflict are not scheduled for their lunch meetings on the same day). The types of the variables are optional; if omitted, type inference will derive them. In the case at hand, the type of the variables a and b can be derived from the signature of $conflict$ and $mtng\_day$.

```
theory T:V{
    ! a[team] b[team]: conflict(a,b) =>
                       mtng_day(a) ~= mtng_day(b).
}
```

Figure 10: An IDP3 theory over the vocabulary of Figure 7 (using correspondences from Figure 9).

For solving problems, the IDP3 system offers a procedural interface in Lua[3] and executes the procedure $main()$. In $main()$, the IDP3 user can overwrite default values of solver options, can invoke Lua functions provided by the IDP3 designers as well as standard Lua functions, and she can also invoke functions written by herself in the procedural component. The information components (vocabularies, structures and theories) are first-class citizens in the Lua code and can be passed as parameters to various functions. The procedural component in Figure 11 illustrates some common use patterns of the Lua interface. In this code, $modelexpand$, $printmodels$ and $allmodels$ are Lua functions provided with the IDP3 system, while # is the Lua operator returning the length of a Lua sequence and $print$ is the standard Lua printing function.

The $modelexpand$ function invokes model expansion inference on a theory T and a structure S (both referring to the same vocabulary), and returns a (possibly empty) Lua sequence of models of T that extend S; by default, $modelexpand$ is bound to return a single model. To obtain more models, another bound can be set with an assignment to the $stdoptions.nbmodels$ option. The $printmodels$

[3]Lua is a scripting language (Ierusalimschy, Henrique de Figueiredo, and Celes 1996) available at `http://www.lua.org`.

```
procedure main(){
    stdoptions.nbmodels=5
    printmodels(modelexpand(T,S))
    models = allmodels(T,S)
    print(#models)
    print(model[5])
    print(model[980])
}
```

This procedure applies model expansion inference on our theory T and initial structure S in two different ways. The first line sets the bound on the number of models to 5. The second line invokes model expansion and prints the sequence of 5 models. The third line also invokes model expansion but returns the sequence of all models and assigns it to the Lua variable $models$. The fourth line prints the number of models in the sequence, the next line prints the fifth model and the final line prints nil as there are only 960 models.

Figure 11: Solving the model expansion problem in IDP3.

function prints the number of models in a sequence of models as well as each of its models. To obtain the sequence of all models, one can use the $allmodels$ function. Indexing can be used to select a particular element in a sequence; if the sequence is empty (models do not exist), the special Lua value nil is returned. Models are represented and printed as structure components so that they can serve as IDP3 input.

Definitions, Aggregates and Optimization

So far, we have seen two extensions of first order logic that are available in IDP3: *types* and *partial* functions (a typed function is partial as it is only defined for the values determined by the types in the signature). Other important extensions are *aggregates* and *definitions*. To illustrate them, we elaborate on our example; at the same time we also introduce another reasoning task, *optimization inference*. The IDP3 code for the extended example is shown in Figure 12.

In the extended scenario, we are concerned with the workload of the company cafeteria where the meetings take place. We introduce the concept of $quiet\_day$ that we *define* as a day in which at most one team holds its meeting. As stated before, definition expressions in FO(ID) are modeled after the way definitions are expressed in text. They define one or more predicate or function symbol in terms of a set of *parameter symbols*. E.g., the concept $quiet\_day$ is defined in terms of the function $mtng\_day$ which we call a parameter of the definition. To distinguish a definition expression from FO sentences, it is written as a set of rules placed between "**define** {" and "}"[4].

Each rule expresses one (base or inductive) case. Head and body of the rule are separated by "<−", called the *definitional* implication to distinguish it from the material implication => (both given in the IDP syntax). The head is an atomic formula of one of the defined predicates and the body can be any formula in first order logic. In contrast with logic programming, variables are explicitly quantified.

[4]The keyword **define** is optional.

```
vocabulary V{
    ...
    type number isa nat
    quiet_day(day)
    nmbr_mtngs(day):number
}
structure S:V{
    ...
    number={0..10}
}
theory T:V{
    ...
    define {
     ! d: quiet_day(d) <− ∼(? t1 t2: t1 ∼= t2 &
                mtng_day(t1)=mtng_day(t2)=d).
    }
    define {
     ! d: nmbr_mtngs(d)=#{tm: mtng_day(tm)=d}.
    }
}
term m:V{
    max{d[day] : true : nmbr_mtngs(d)}
}
procedure main(){
    stdoptions.cpsupport=true
    models, optimal, cost = minimize(T,S,m)
    print(models[1])
    print(optimal)
    print(cost)
}
```

Figure 12: The running example extended to illustrate aggregates, definitions and optimization. The missing lines of code should be taken from Figures 7, 8, and 10.

To give such formal rule sets the intuitive reading of definitions in mathematics, the semantics chosen for them is an extension of the well-founded semantics (Van Gelder, Ross, and Schlipf 1991; Denecker and Ternovska 2008), because the well-founded semantics correctly formalizes the most common forms of definitions found in text (Denecker and Vennekens 2014). The IDP3 definition given in Figure 12 formally expresses the intended meaning for the concept $quiet\_day$: a day d is a quiet day if it is not the case that two different teams $(t1, t2)$ have their meeting day on d.

Similarly, we include in the vocabulary a function $nmbr\_mtngs$ that we want to define as the function that maps a day to the number of teams meeting on this day. The new symbol ranges over the new type $number$, which we introduce in the vocabulary as a subtype of the natural numbers (a built-in type *nat*) and specify in the structure as the set of numbers from 0 to 10. As the relation $quiet\_day$, also this function is defined in terms of the parameter $mtng\_day$. In general, functions are defined by sets of rules of the form $f(t_1, \ldots, t_n) = t \leftarrow body$. In the case at hand, the body degenerates to "true" and is omitted. Importantly, the function value here is given by the *cardinality aggregate* $\#\{tm : mtng\_day(tm) = d\}$. This aggregate represents the cardinality of the set $\{(tm) \mid mtng\_day(tm) = d\}$, i.e., the number of teams meeting on day d. Besides cardinal-

ity, IDP3 also supports *minimum*, *maximum*, *sum* and *product* aggregates. They have a slightly different syntax. An overview of the supported aggregates is given in Figure 13.

Assume that the manager of the cafeteria wishes to minimize the maximal workload for the cafeteria. To solve this problem, another form of inference is needed called *optimization* inference. This is done by the procedure call $minimize(T, S, m)$ in the $main()$ procedure of Figure 12. The procedure call contains yet another sort of component of IDP3: the *term* component. Its role is to give a name to a term. Referred to by its name, the term can then be used inside Lua procedures. Here, the term of interest is the maximum n in the set of pairs (d, n) defined as $\{(d, n) \mid n = nmbr\_mtngs(d)\}$. This is a simple *maximum* aggregate in which there are no extra conditions on d. According to the translation table of Figure 13, its IDP3 syntax is $max\{d[day] : true : nmbr\_mtngs(d)\}$. The middle part "$true$" is the trivially true extra condition on the selected values for d. The optimization inference performs a search for a model that minimizes the value of the term referred to by m. The call not only returns a model, but also whether optimality could be shown and the value of the term. So, minimization is on the maximal number of meetings on the same day. In other words, the call $minimize(T, S, m)$ returns a schedule that minimizes the maximum number of lunch meetings scheduled for a single day (informally, it offers a "balanced" schedule). The grounder of the IDP3 system is unable to derive a bound on the value of the optimization term m. To avoid an infinite grounding, the option $cpsupport$ must be on.

FO	IDP3
$\#\{\overline{x} : F\}$	#{x1 ... xn : F}
$sum\{(\overline{x}, t) : F\}$	sum{x1 ... xn : F : t}
$prod\{(\overline{x}, t) : F\}$	prod{x1 ... xn : F : t}
$max\{(\overline{x}, t) : F\}$	max{x1 ... xn : F : t}
$min\{(\overline{x}, t) : F\}$	min{x1 ... xn : F : t}

Figure 13: Translation table for aggregates.

Partial Information and Constructed Types

As an alternative elaboration of our example, assume it is decided that team $T1$ meets on Monday ("is certainly true") and team $T2$ does not meet on Tuesday ("is certainly false"). This partial knowledge can be expressed in the structure as shown in Figure 14 (we note the use of markers <ct> and <cf>).

```
structure S:V{
    ...
    mtng_day<ct> = {T1 ->M}
    mtng_day<cf> = {T2 ->Tu}
}
```

Figure 14: Partial knowledge in a structure.

Alternatively, we may want to express this information in the theory. However, in the theory we can only express information about domain elements if we have symbols in the vocabulary to refer to them. Hence, we need to extend the vocabulary with constants $mon, tue, \ldots, t1, t2, \ldots$ to denote days and teams; furthermore, the structure needs to be extended to specify the interpretation for the new constants by means of the statements $mon = M, \ldots, t1 = T1, \ldots$. Only then we can express constraints such as $mtng\_day(t1) = mon$ or $mtng\_day(t2)\sim=tue$ in the theory. This verbose way is a consequence of the fact that functions and constants are not limited to their Herbrand interpretation as in ASP and Prolog. A shortcut is to make use of *constructed types* to enforce Herbrand interpretations over certain types. Figure 15 shows how constructed types impose the same constraints on the function $mtng\_day$. As the domain of these types is fixed in the vocabulary, they are not part of any structure.

```
vocabulary V{
    type day constructed from {M,Tu,W,Th,F}
    type team constructed from {T1,T2,T3,T4,T5}
    ...
}
theory T:V{
    ...
    mtng_day(T1) = M.
    mtng_day(T2) ~= Tu.
}
```

Figure 15: Constructed types.

More about Definitions

The definitions we discussed above are simple and can be expressed in first order logic as equivalences. For example, the equivalence $!d : quiet\_day(d) <=> \sim(?t1\ t2 : t1\sim=t2\ \&\ mtng\_day(t1) = mtng\_day(t1) = d)$ correctly expresses the definition of $quiet\_day$. While this works for all non-inductive definitions, it is well known that inductive definitions in general cannot be expressed through FO equivalences.

Definitions are the most substantial extension that IDP3 offers with respect to first order logic. Not only do they offer the designer a facility to define concepts, they also increase the expressiveness. The archetypal example of a relation that cannot be expressed in first order logic is the transitive closure of the edges in a graph. The inductive definition of this relation, say T, for a graph (N, E) with nodes N and edges E is often stated as follows:

- If $(a, b) \in E$, then $(a, b) \in T$,
- If for some $c \in N$, it holds that $(a, c) \in T$ and $(c, b) \in T$, then also $(a, b) \in T$.

In IDP3, we can model it as in Figure 16.

To further illustrate the power of definitions, we present in Figure 17 a representation of a simple graph problem that requires selecting edges among nodes so that in the resulting graph all vertices are reachable from a (given) node *root* and none of (given) forbidden edges are selected. The main difficulty is that the set of vertices reachable from the root is not

```
vocabulary V{
    type node
    edge(node,node)
    trans(node,node)
}
structure S:V{
    edge={...}
}
theory T:V{
    define {
     ! x y: trans(x,y) <- edge(x,y).
     ! x y: trans(x,y) <- ? z: trans(x,z) & trans(z,y).
    }
}
```

Figure 16: Transitive closure of the edges in a graph.

expressible in first order logic. To overcome this problem, we introduce the auxiliary unary predicate symbol *reachable* and express it through the inductive definition provided in Figure 17. Additional axioms express that the defined relation *reachable* is the set of all nodes and no edges are forbidden. An interesting aspect is that here, the defined relation *reachable* is known initially while the parameter *edge* in terms of which it is defined is unknown. Hence, IDP3 searches for an interpretation of the parameter *edge* such that the defined relation *reachable* has the given value. This sort of input/output pattern is different from that of Prolog and DATALOG systems, and it shows the declarative nature of definitions. It is a powerful aspect of IDP3 as well as ASP systems.

In the example in Figure 17, one can check that the edges (A, D) and (D, C) must appear in every solution for the relation *edge*. Also, at least one of (D, B) or (C, B) must be present. Other allowed edges are not constrained. Thus, one possible value for *edge* is $\{(A, D), (D, B), (D, C)\}$ and another one is $\{(A, D), (C, B), (D, C), (B, D)\}$.[5]

FO(ID) (IDP3) and ASP

On the conceptual level, FO(ID) and ASP are quite different. Whereas ASP has its foundation in nonmonotonic and commonsense reasoning, FO(ID) is based on a definition construct inspired by the structure of definitions used in mathematics. Negation in ASP is viewed as a non-classical epistemic or default operator. In FO(ID), it is the definitional rule operator $\leftarrow$ that is non-classical, while negation in the bodies of definition rules is classical. And yet, despite these different foundations, there are strong structural relationships between ASP and FO(ID). On the language level, FO(ID)'s rule-based definition construct resembles ASP rules, and FO axioms resemble ASP constraints. We illustrate these similarities with the problem of finding a Hamiltonian cycle in a directed graph. An answer set program encoding the problem is shown in Figure 18.

That program has a typical structure resulting from

[5]This theory can be accessed and experimented with on the IDP-IDE webpage at `http://dtai.cs.kuleuven.be/krr/idp-ide/?present=forbidden`.

```
vocabulary V{
    type node
    forbidden(node,node)
    edge(node,node)
    reachable(node)
    root:node
}
structure S:V{
    node = {A..D} // a shorthand for {A; B; C; D}
    forbidden = {(A,A); (A,B); (A,C); (B,A);
            (B,B); (B,C); (C,C); (C,D); (D,D)}
    root = A
}
theory T:V{
    // inductive definition of reachable
   define {
    reachable(root).
    ! x: reachable(x) <- ? y: reachable(y) & edge(y,x).
   }
    // The graph is fully connected
    ! x: reachable(x).
    // No forbidden edges
    ! x y: edge(x,y) => ~forbidden(x,y).
}
```

Figure 17: A graph problem.

generate	$\{In(x,y)\}$<-$Edge(x,y)$.
define	$T(x,y)$<-$In(x,y)$. $T(x,y)$<-$T(x,z), T(z,y)$.
test	<-$In(x,y), In(x,z), y \neq z$. <-$In(x,z), In(y,z), x \neq y$. <-$Node(x), Node(y), not\ T(x,y)$.

Figure 18: A generate-define-test ASP program encoding the existence of a Hamiltonian cycle problem.

following the generate-define-test (GDT) methodology (Lifschitz 2002) (discussed in this issue by Faber, Gebser and Schaub (2016)). This methodology leads to three sorts of modules. The first of them *generates* the space of candidate solutions (in our example, the space of all subsets of the set of edges of the input graph; they are possible instantiations of a relation In). The generate module commonly relies on the construct of choice rules (as in our example) or, alternatively, uses disjunctive rules. The second one *defines* some additional concepts that are useful in identifying solutions (here, the transitive closure of the relation In). Finally, the third one specifies constraints of the problem. These constraints narrow down the space of candidate solutions to those that represent the valid ones (here, the constraints ensure that exactly one edge comes into each node, exactly one edge leaves each node and, finally, that all nodes are connected to each other both ways; that last condition requires an auxiliary concept of the transitive closure). The horizontal lines in Figure 18 make this structure explicit.

The corresponding IDP3 solution to the problem has a similar format. We present its theory component in Figure

19.

```
theory T:V{
    ! x y: In(x,y) => Edge(x,y).
    ---------------------------------
    define {
      ! x y: T(x,y) <- In(x,y).
      ! x y z T(x,y) <- T(x,z) & T(z,y).
      }
    ---------------------------------
    ! x y z: ~(In(x,y) & In(x,z) & y~= z).
    ! x y z: ~(In(x,z) & In(y,z) & x~= y).
    ! x y: ~(Node(x) & Node(y) & ~T(x,y)).
}
```

Figure 19: An IDP3 theory encoding the existence of a Hamiltonian cycle problem.

The similarity is striking. The first sentence plays the role of the *generate* module in the program in Figure 18. The definition of the transitive closure mirrors the *define* module. Finally, the last three sentences are the three constraints of the *test* module cast in the IDP3 syntax. As an aside, we note that a direct translation from natural language to the IDP3 syntax of these constraints would more likely be as in Figure 20.

```
    ! x y z: In(x,y) & In(x,z) =>  y = z.
    ! x y z: In(x,z) & In(y,z) =>  x = y.
    ! x y: T(x,y).
```

Figure 20: A direct IDP3 representation of the test constraints.

Almost all GDT programs can be translated into IDP3 following the idea outlined above. The encoding of the *generate* module does not require any special syntax. In fact, in many cases the *generate* part of a GDT program disappears entirely from the corresponding IDP3 theory. The converse is also true. A large class of IDP3 theories allows for automated rewritings into the language of ASP. The key in such translations is to properly construct the choice rules to "open" some of the predicates.

Similarities between ASP and FO(ID) can be found not only in the structure of programs (theories). On the system level, the core of IDP3 is a model generator that is developed using similar technologies as current ASP solvers, and offers similar functionalities.

Concluding Remarks

In this paper, we focused on the model-generation task because of its natural applications in solving search and optimization problems. This is also the focus of ASP and ASP implementations. We noted that model generation can be implemented for other logics. We mentioned some of them and then described in detail the logic FO(ID) and the associated reasoning system IDP3.

However, it is important to point out that the knowledge present in both FO(ID) theories as well as in answer set programs can support many other reasoning tasks besides model generation. That observation has played a central role in the development of the system IDP3 and is reflected in its functionality (cf. Figure 6). Similarly, it underlined some developments in ASP (cf. the paper by Kaufmann et al. (2016) in this issue). In particular, most implementations of ASP support skeptical and brave reasoning, and add-ons facilitating abduction and planning were developed for some systems, as well (Eiter et al. 2003; 1999).

The field of computational logic has an urgent need for integrative frameworks that recognize that many reasoning tasks are needed in knowledge-intensive applications and that these tasks can all be driven by a single well-designed underlying knowledge base. Formalisms and systems discussed in this special issue are on the intersection of several related lines of research, building on the advances in classical logic, automated reasoning, logic programming, databases, satisfiability, satisfiability modulo theories, constraint programming, fixpoint logics, and description logics. As such, they are well suited to play this integrative role. Their modeling capabilities which, in important respects such as the ability to capture inductive definitions, go beyond SAT/CSP formalisms, as well as the computational effectiveness of their reasoning software demonstrate that. We posit that developing FO(ID), ASP and related formalisms with this goal in mind is essential both for the theory of logic-based computation and for practical applications.

Acknowledgments

This research was supported by the project GOA 13/010 Research Fund KULeuven and projects G.0489.10, G.0357.12, and G.0922.13 of the Research Foundation - Flanders.

References

Aavani, A.; Wu, X. N.; Tasharrofi, S.; Ternovska, E.; and Mitchell, D. G. 2012. Enfragmo: A system for modelling and solving search problems with logic. In Bjørner, N., and Voronkov, A., eds., *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2012*, volume 7180 of *LNCS*, 15–22. Springer.

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases*. Addison-Wesley.

Aho, A. V., and Ullman, J. D. 1979. The universality of data retrieval languages. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, 110–120.

Apt, K.; Blair, H.; and Walker, A. 1988. Towards a theory of declarative knowledge. In Minker, J., ed., *Foundations of Deductive Databases and Logic Programming*, 89–142. Morgan Kaufmann.

Cadoli, M.; Ianni, G.; Palopoli, L.; Schaerf, A.; and Vasile, D. 2000. NP-SPEC: an executable specification language for solving all problems in NP. *Comput. Lang.* 26(2-4):165–195.

Clark, K. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logic and data bases*. New York-London: Plenum Press. 293–322.

De Cat, B.; Bogaerts, B.; Bruynooghe, M.; and Denecker, M. 2014. Predicate logic as a modelling language: The IDP system. *CoRR* abs/1401.6312.

Denecker, M., and Ternovska, E. 2008. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log.* 9(2):14:1–14:52.

Denecker, M., and Vennekens, J. 2014. The well-founded semantics is the principle of inductive definition, revisited. In Baral, C.; Giacomo, G. D.; and Eiter, T., eds., *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning, KR 2014*. AAAI Press.

Denecker, M. 1998. The well-founded semantics is the principle of inductive definition. In Dix, J.; del Cerro, L. F.; and Furbach, U., eds., *Proceedings of the European Workshop on Logics in Artificial Intelligence, JELIA 1998*, volume 1489 of *LNCS*, 1–16. Springer.

Denecker, M. 2000. Extending classical logic with inductive definitions. In Lloyd, J. W.; Dahl, V.; Furbach, U.; Kerber, M.; Lau, K.-K.; Palamidessi, C.; Pereira, L. M.; Sagiv, Y.; and Stuckey, P. J., eds., *Computational Logic, CL 2000*, volume 1861 of *LNCS*, 703–717. Springer.

East, D., and Truszczynski, M. 2006. Predicate-calculus-based logics for modeling and solving search problems. *ACM Trans. Comput. Log.* 7(1):38–83.

Eiter, T.; Faber, W.; Leone, N.; and Pfeifer, G. 1999. The diagnosis frontend of the dlv system. *AI Commun.* 12(1-2):99–111.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2003. A logic programming approach to knowledge-state planning, II: the dlv$^{\mathrm{k}}$ system. *Artif. Intell.* 144(1-2):157–211.

Faber, W.; Gebser, M.; and Schaub, T. 2016. Modeling and language extensions. *AI Magazine*. This issue.

Ierusalimschy, R.; Henrique de Figueiredo, L.; and Celes, W. 1996. Lua – an extensible extension language. *Software: Practice and Experience* 26(6):635–652.

Kaufmann, B.; Leone, N.; Perri, S.; and Schaub, T. 2016. Grounding and solving in answer set programming. *AI Magazine*. This issue.

Lifschitz, V. 2016. Answer sets and the language of answer set programming. *AI Magazine*. This issue.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence* 138:39–54.

Reiter, R. 1980. A logic for default reasoning. *Artificial Intelligence* 13(1-2):81–132.

Van Gelder, A.; Ross, K. A.; and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *J. ACM* 38(3):620–650.